The 6th Joint International Conference on Multibody System Dynamics and
The 10th Asian Conference on Multibody System Dynamics
November 1-5, 2020, New Delhi, India

Exudyn – A C++ based Python package for flexible multibody systems

Johannes Gerstmayr*

* Department of Mechatronics
University of Innsbruck
Technikerstr. 13, 6020 Innsbruck, Austria
johannes.gerstmayr@uibk.ac.at

Abstract

The present contribution introduces the design, methods and capabilities of the open source multibody dynamics simulation code Exudyn, which has been developed at the University of Innsbruck since 2019. The code has been designed for rigid and flexible multibody systems, with focus on performance for regular multi-core CPUs, with script language based modeling and for prior use in science and education, but also in industry. The open source code consists of a main C++ core, a rich Python interface including pre- and post-processing modules in Python, and a collection of rigid and flexible bodies with according joint, load and sensor functionality. Integrated solvers allow explicit and implicit time integration, static solution, eigenvalue analysis and optimization. In the paper, the code design, structure, computational core, computational objects and the multibody formulations are addressed. In addition, the computational performance is evaluated with examples of rigid and flexible multibody systems. The results show the significant effect of multithreading even for comparatively small systems as well as for larger models.

1 Motivation

The idea to write an open source C++ code for multibody system dynamics simulation is not new at all. There are highly efficient multibody dynamics codes, and recently new codes have evolved in the computer graphics and robotics areas [1, 15]. However, these codes are specialized to rigid bodies and contact, and usually based on kinematic trees, using a minimum coordinates formulation. As with flexible bodies or compliant joints, minimum coordinates are not always (computationally) advantageous, and extension of such existing, specialized codes is very time consuming – it may in the end be necessary to rewrite such a code from scratch. Furthermore, maximum computational efficiency, which can be achieved on GPUs nowadays, may not always be required for practical applications as compared to code extensibility or an intuitive user interface. A large amount of time is usually spent to create and to check the correctness of multibody models, which is why the user interface – no matter whether it is graphical or text-based – deserves high importance. Finally, flexible bodies may be as complicated as higher order nonlinear rods or flexible bodies with model order reduction, leading to very different sizes and complexities of computational objects, having single degrees of freedom (DOF) masses versus flexible bodies with more than 100 flexible modes. Thus, flexible bodies are much harder to integrate into existing environments that are based on rigid bodies and kinematic trees.

The motivation for a new code therefore relied on the extensibility, while still achieving sufficient performance. The idea is to use efficient and modern implementation techniques only when absolutely necessary. At the remaining places, the code shall be simple, readable and extensible and user interaction shall be as simple and safe as possible. Therefore, there are nearly no external libraries that are heavily used in the present code, as they would dictate parts of the structure or of the interface. Furthermore, while most open source projects are missing a full and up-to-date documentation of interfaces as well as the theory behind the objects, the documentation is an integrated part of Exudyn, which is continuously developed together with the code.

There are powerful open source codes that include already flexible bodies, such as Projectchrono[1] [10], MBDyn [9], or Hotint [2]. However, first tests showed that an adaptation of such a big project to a fully new design would have implicated to rewrite such a code, while it still would not meet design goals such as the integrated documentation of added objects. Finally, even that Exudyn has been started from scratch, some of the existing linear algebra and helper routines, which were also used in Hotint, have been revised and adapted to the new project.

[1] `https://projectchrono.org`

2 Introduction to Exudyn

Exudyn – FlEXible mUltibody DYNamics – can be understood as the successor of the previously developed multibody code HOTINT – High Order Time INTegration [2], which was reaching significant limitations of its extension due to the original design as a time integrator, but not as a multibody code. Exudyn [4] has been planned as a research and education multibody code, with a thorough design for the integration of a scripting language, efficient solvers, equation building, graphics visualization and integrated documentation. As compared to comparable multibody codes, the Python scripting language has not been attached, but the code design is built upon the existence of such a scripting language, which reduces code duplication significantly.

The underlying C++ code focuses on a implementation-friendly but otherwise efficient design.Important parts of the code are parallelized using specialized multithreading approaches. The Python interface is small and mostly generated automatically, but with lots of parameter checking such that the building of models and error finding is convenient for inexperienced users. The access of objects via Python is therefore restricted, because the full exposure of computational objects to Python would require too many checks of valid arguments. Exudyn is hosted on github [4] which includes all sources, examples and documentation. Pre-compiled versions can be obtained conveniently via pypi, using the simple Python environment command `pip install exudyn`.

Exudyn version 1.3.75.dev1 – including tests – currently roughly consists of 157.000 lines, of which are 67% C++ code, 12% Python modules code, 10% definition and code generation and 11% tests. Furthermore, it should be noted that 40% of the C++ code is automatically generated. The automatic code generation not only alleviates consistency between Python and C++ interfaces, but also creates the reference manual. As a consequence, the definition of quantities, such as the stiffness of a spring-damper, is provided only once, while this information is passed to C++ and Python interfaces and to the documentation, all of them being always consistent. On average, only 200 lines of manual code are needed per item, such as a rigid body or a marker, which is important for code maintainability and for the realization of larger code changes. The current documentation [4] covers more than 700 pages, including installation, tutorial, description of entry points and interfaces, theory and reference manual for every of the more than 80 items, such as objects or loads. The documentation also includes tracking of changes and bugs, which allows users to immediately discover new features.

Coupling between C++ and Python is realized with the header file based library pybind11 [7], which enables a rich Python interface, and small code size. Using the Python setuptools, Exudyn can be built for Windows, Linux and MacOS (although limited), and has even been compiled for ARM-based Raspberry Pi. Instead of starting with the definition of nodes, objects and markers, there exist more than 100 examples and test models, which can be taken as a starting point to create new models. Furthermore, there are tutorials in the documentation [4], as well as tutorial videos on youtube. Possible use cases range from 1-DOF-oscillators to 1 000 000 DOF finite element or particles simulations.

While the core part of the code is written just by one author, many colleagues contributed to tests, formulations, specialized objects and examples, which are mentioned in detail on github [4]. The paper and presentation will show details on the implementation, formulation, performance, code usage and examples as well as test results. The description given here can only be a brief introduction into the code, omitting significant parts of the functionality. The full and continuously updated documentation, from which some figures and text fragments are taken, is available on github[2].

2.1 Design of Exudyn

The design goal of Exudyn has been a comparatively light-weight multibody dynamics software package for research, education and limited industrial applications. Linking to other open source projects shall be possible via Python. It shall include basic multibody dynamics components, such as rigid bodies and point masses, flexible bodies, joints and contact. Models shall be scripted in Python, either within a list-like collection of bodies or joints, or a highly parameterized model using for-loops and external libraries. Additional Python utility modules shall enable convenient building and post processing of multibody models, shifting implementation of complex but not performance critical functionality to Python. It shall further include explicit and implicit dynamic solvers, static and eigenvalue solvers as well as parameter variation and optimization methods. It shall be designed to be able to both efficiently simulate small

[2] https://github.com/jgerstmayr/EXUDYN/blob/master/docs/theDoc/theDoc.pdf

scale systems and large scale systems with $n_{DOF} < 1\,000\,000$. It shall include a 3D visualization in order to immediately detect modeling errors, to allow user defined objects and solvers in C++ and in Python. The system core shall be a redundant multibody dynamics formalism with constraints, but also including kinematic trees for efficient modeling of rigid body tree structures (which may be solved with explicit solvers). Parallelization and vectorization shall be enabled through multithreaded parallelization.
There are also some non-goals of Exudyn:

- currently, it is not designed to run simulations on GPU;
- currently, there is no graphical user interface for creation of multibody models planned.

As mentioned in Section 6, the creation of models as well as the start and evaluation of simulations is done using scripts entirely written in Python. Thus, the C++ module in the background acts as a state machine, which is driven by the user's script. A larger set of Python functions allows the user to understand the current 'state' of the system, not only of the coordinates but also of the total system setup. A dense network of safety checks is integrated in order to allow the user to distinguish illegal input from system errors (which may occur, e.g., if modules are not fully completed).

3 Structure of the code

The code consists of a C++ core module denoted as exudynCPP and a collection of Python utility modules, see Fig. 1.

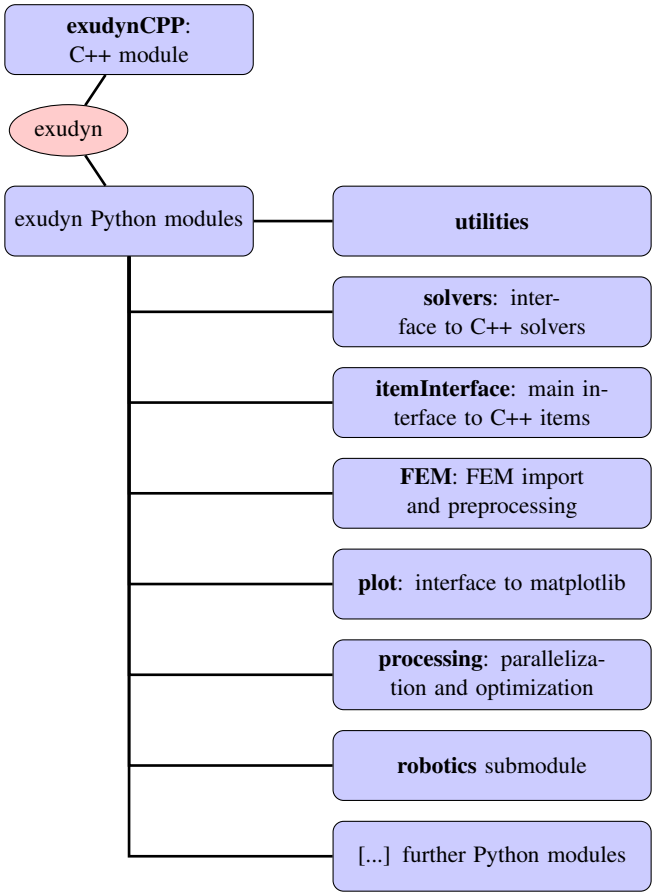

Figure 1: Overview of Exudyn modules.

3.1 C++ core module

The C++ core is (currently) exclusively[3] linked to Python using the C++ tool pybind11 [7]. Due to the fact that pybind11 is highly templated, which can cause large compilation costs[4] and impedes conventional

[3]Linking to other high level languages such as Julia may be realized in future with limited effort, as most Python interfaces are created automatically; currently, there is no straight-forward way to use the C++ code without Python.

[4]Due to specific measures to reduce compilation costs of Exudyn, an Intel Core-i9 with 14 cores (28 threads) is able to perform multithreaded compilation and linking of Exudyn V1.3.75.dev1 in only 74 seconds. The main C++ implementation

debugging, the interface between Python and C++ is reduced to very few core parts, see Fig. 2.
The C++ core parts include only few structures, such as simulation settings, visualization settings, solvers and sparse matrices, which are needed for efficient interaction with Exudyn. Larger efforts have been put on the development of these parts in order to allow the user to easily detect errors in the scripting of models, see the examples section. For this reason, there is a layer between the creation of items, e.g., a rigid body, marker or sensor using conventional Python dictionaries. These dictionaries contain the whole data for a single item, allowing to write as well as read this data. This data is checked before transferring it to the C++ module, which otherwise would lead to hard to interpret exceptions. Furthermore, as all items are transferred between C++ and Python by a single interface, the change of the interface as well as the implementation overhead is greatly reduced.

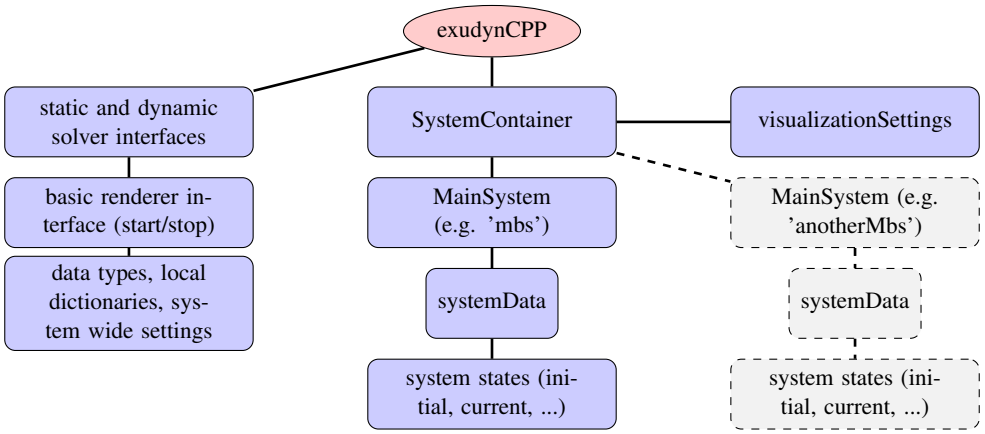

Figure 2: Structure of Exudyn C++ module.

3.2 Python modules

C++ code is computationally more efficient than pure Python code. However, maintainability of efficient C++ code is time consuming and, in general, error prone. Many parts of a multibody code, regarding model setup and validation, pre- as well as postprocessing are not performance critical. For this reason, these parts are shifted to Python modules. These Python modules, see Fig. 1, are designed to make scripting as easy as possible and to reduce the C++ parts. Python modules include the interface (`exudyn.itemInterface`) between Python items and dictionaries transferred to the C++ core, utilities for optimization and parameter variation (`exudyn.processing`), a plotting interface to matplotlib (`exudyn.plot`), interfaces for rigid body dynamics (`exudyn.rigidBodyUtilities`) and for finite element models (`exudyn.FEM`). Besides further modules for artificial intelligence, beams and graphics representation, there is also a robotics submodule (`exudyn.robotics`) which can be used to easily set up serial or mobile robots.

3.3 Download, installation and building

Since more than two years, Exudyn is hosted on github [4]. The source code, which can be downloaded and compiled, contains the C++ code, setup files for easy compilation, the Python modules as well as the documentation (`theDoc.pdf`) including its latex source files. Since Exudyn Version 1.2.0, it also has been added to the pypi index, see https://pypi.org/project/exudyn. This allows to easily install pre-compiled wheels on many platforms, such as Windows or Linux using the simple command `pip install exudyn`. Pre-release versions can be fetched using `pip install exudyn -pre`.
Building[5] Exudyn is enabled with Python's `setuptools`. By going to the main directory `exudyn/main`, the Python file `setup.py` includes all information to build Exudyn on many platforms using the command

```
python setup.py install
```

files, which include the pybind11 interface, need almost all of the compilation time of a single thread, while the remaining threads perform compilation of the remaining 120 implementation files.

[5]Compilation and linking of all C++ parts which are collected in a Python wheel together with all Exudyn Python libaries.

Building has been tested at least for Windows 10, Ubuntu 18.04 and 20.04, MacOS as well as for RaspberryPi 4b. However, MacOS requires often some adaptations, see `theDoc.pdf`.

4 Basic structure of the C++ core

The C++ core consists of a `MainSystem` which is used to add, modify and read items (see Section 4.1) in the system. Items are created by a typical object factory, which creates a C++ item from the Python dictionary definition. For inter-operation between C++ and Python, every item has functions to read and write data structures.
For future extensibility and performance reasons, the items are separated into three parts:

- **main** (M) part: This is the top-level structure, which includes the interface to Python and further non-performance critical parts. Python linking leads to higher compilation times, which is why this part is only included in very few places. The main part also includes pre-assembly checks.
- **computational** (C) part: this is the main, performance critical part. It is kept light-weight and is reduced to essential parts that are needed during computation. Focus is put on conciseness, reducing number of lines for easy readability and avoidance of programming errors. Typically, objects include only a few dozens up to 200 lines of code.
- **visualization** (V) part: This part is only related to visualization and is thus decoupled from computation. Visualization parts are only linked to graphics related code, which is excluded if the code is compiled without the graphics renderer.

In addition to these parts, data structures for item parameters, such as the mass of a rigid body, are put into a separate `parameter` structure, which clearly separates data and computational parts. This should alleviate portability, especially to GPU-like hardware, in the future.

4.1 Computational items

The computational items are separated into few conceptually different components, namely nodes, objects, markers, loads and sensors. Separation into these groups makes implementation and extension easier and allows to implement various formulations. In additional to computational properties, items can be created and modified via the system they belong to. Furthermore, items can be visualized in the 3D rendering window. While nearly every item possess the ability for some standard graphical representation, bodies can be enriched with conventional 3D graphics imported by STL[6] meshes. Naturally, all items are represented on every layer (M, C, V) by means of C++ classes, together with the corresponding Python interface class.
Markers provide interfaces on coordinate, position or orientation level that can be also used by loads, such as forces or torques. The separation of nodes and objects allows to use a set of rigid body nodes with different formulations for rigid or flexible bodies, leading to no additional implementation efforts. Joints are provided in index 3 and index 2 version, which allows to use solvers with index reduction and a convenient computation of initial accelerations. Besides the rather independent implementation of objects or nodes, there is a specialized contact module which realizes an optimized contact search as well as evaluation of contact forces and Jacobians using multithreading. Many items allow to integrate Python user functions, which break down performance, but allow nearly unrestricted interaction and couplings inside but also outside of Python (e.g., TCP connection with MATLAB). While it is recommended to simulate large-scale particle systems with one of the many excellent simulation codes such as Projectchrono [10], Exudyn is also capable of simulating more than 1 million rigid bodies without special hardware or installation.

4.1.1 Nodes

Nodes provide the coordinates (and the degrees of freedom) to the system. They have no mass, stiffness or other physical properties assigned. They can represent coordinates (of different types, see Section 4.2.3), such as the position of a mass point, or generalized coordinates for minimal coordinates formulation.
Nodal coordinates and its assigned kinematical quantities (such as a rotation matrix) can be measured. Nodes can be used by a single object (e.g., rigid body), or shared by several objects, such as with finite

[6] Stereolithography file format; also known as Standard Triangle Language

elements. For the system to be solvable, equations need to be assigned for every nodal coordinate (which represents a system state).

4.1.2 Objects

Equations are provided by (computational) objects. Objects need one or more according nodes, which provide the respective coordinates, such as a position node for a mass point. In the special case of algebraic constraints, the algebraic variables are automatically assigned to the constraints (without nodes), as a double usage of these coordinates is not considered meaningful.
Objects may provide derivatives and have measurable quantities (e.g., displacements or rotations) and they provide position or rotation Jacobians, which are used to apply, e.g., forces. While objects provide equations to the residual, bodies also provide parts of the mass matrix and they have geometrical extension. When adding mass, equations are equivalent if one adds a single mass point with mass m to a point node, or if two mass points with mass $m/2$ are added to the same point node. The same is true for a spring-damper, if adding one spring with stiffness k or two springs with stiffness $k/2$ to the same markers leads to identical behavior (except for numerical round-off errors).
Objects can be as various as follows:

- **generic object**, e.g. represented by generic second order differential equations
- **body**: has a mass or mass distribution; markers can be placed on bodies; loads can be applied; constraints can be attached via markers; bodies can be:
  - **ground object**: has no nodes
  - **single-noded body**: has one node (e.g. mass point, rigid body)
  - **multi-noded body**: has more than one node; e.g., **finite element**
- **connector**: uses markers to connect nodes and/or bodies; adds additional terms to system equations either based on stiffness/damping or with constraints (and Lagrange multipliers). Possible connectors:
  - **algebraic constraint** (e.g. constrain two coordinates: $q_1 = q_2$)
  - **classical joint**
  - **spring-damper** or penalty constraint

4.1.3 Markers

Markers are interfaces for objects or nodes and are used by connectors and loads. Connectors and loads cannot act directly on nodes or objects without markers. Markers can represent points with position Jacobian, a rigid body frame with position and rotation Jacobians or some generalized coordinate(s). As a benefit, connectors only need to be designed for usage with specific markers, such as a rigid body marker, without knowing the exact behavior of the underlying objects, and with no difference if applied to a node or body.
Typical system connectivities are shown in the system graph of a simple double pendulum, see Fig. 3.

4.1.4 Loads

Loads are used to apply (generalized) forces and torques to the system. Loads contain constant values, which are applied as a step function at the beginning of a dynamic simulation. If a static computation is performed prior to the dynamic simulation, it starts from the static equilibrium. The otherwise constant load values may be varied by means of Python user functions, such that they are evaluated correctly in higher order time integration methods.

4.1.5 Sensors

Sensors are only used to measure item quantities in order to create requested output quantities. They can be used to create closed loop systems or they can be easily displayed using a `PlotSensor(...)` utility function, which provides an interface to Python's `matplotlib`.

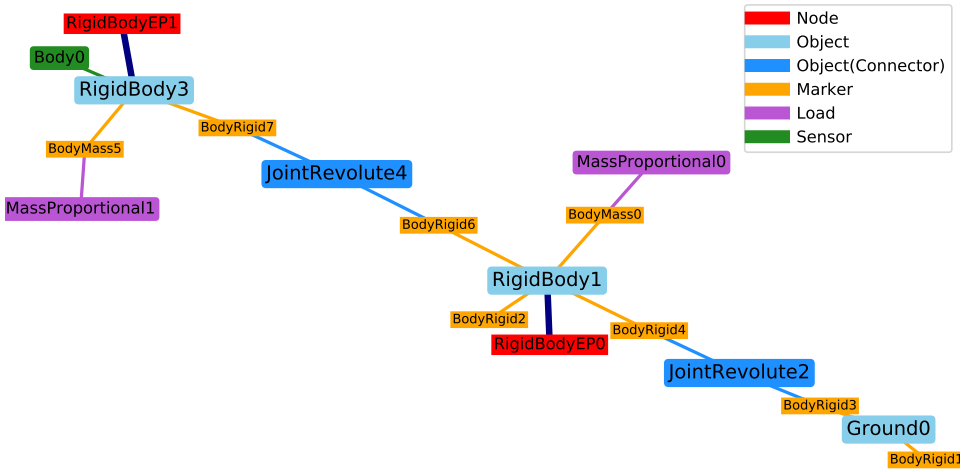

Figure 3: System graph for double pendulum created with Exudyn's `DrawSystemGraph(...)` function. Numbers are always related to the node number, object number, etc.; note that colors are used for nodes, objects, markers, etc. → the label names (which do not include `Object..`, `Node..`) need to be read with the color information!

4.2 System

The (multibody) system is represented by a `MainSystem`, which represents the Python interface to a (computational) `CSystem` with all system-wide functions and a `VisualizationSystem` for all graphics operations, both implemented by C++ classes.
The computational system holds all data inside a data structure (`systemData`), consisting of

- all system states (including some helpers such as accelerations)
- the lists of computational items, such as list of nodes, objects, etc.
- local-to-global coordinate mappings
- several lists of items with specific properties, such as list of bodies and connectors, list of objects with/without user functions, etc., which are used to reduce overheads in certain system functions and to facilitate parallization

System states are available for a list of configurations, mainly,

- current configuration (which is computed during or at end of computational steps)
- reference configuration (specifically needed for finite elements)
- initial configuration
- start-of-step configuration (for solvers)
- visualization configuration (representing the configuration to be rendered)

In specific, reference and initial configurations are distinguished. Reference values represent a specific configuration for finite elements in which their (undeformed) geometry is defined. Reference values are also used by joints in which global joint axes may be defined. On contrary, initial values represent coordinates added to reference values (even for rotational parameters), and thus have a more abstract meaning. They are updated along computational steps and may be stored or loaded to retrieve a specific solution of the system.

4.2.1 Assembling

After the user has created and added all items to a multibody system, some linking and pre-checking of items has to be performed by calling the `Assemble()` function. This function includes sub-functions

- pre-assemble object initialization
- check system integrity

- assign nodal coordinates to global (system) coordinates
- assign object coordinates to global (system) coordinates
- pre-compute item lists (for efficiency)
- initialize system coordinates for initial and reference configuration based on user-defined nodal quantities
- initialize the general contact modules
- post-assemble object initialization

During the system integrity check, a large amount of type and size checks is performed before assembling the model, such that common pitfalls are avoided for the user. These checks are valid ranges of all indices in items, such as node indices in bodies or marker indices in connectors. These checks cannot performed earlier, because in some cases – due to cyclic dependency of objects – some items may be added later to the system than they are referenced by other items. Furthermore, items include a large amount of information, e.g., the valid type of markers for connectors, type of nodes for objects, or available output variables in items. For standard items, all types are checked during the integrity check.
User functions may even change the connectivity of objects or nodes. However, in such cases parts of the assemble procedure needs to be called in order that the system stays consistent.
After the assemble process, the system can be visualized and quantities can be measured.

4.2.2 Computational functions

The computational system is one of the larger parts of the C++ code, implementing widely parallelized system functions which are used during static and dynamic solvers as well as for system linearization. These functions include computation of the system mass matrix, constraint reaction forces, and generalized forces of objects and loads.
Hereby, components of the mass matrix are summed for all objects that are linked to certain nodal coordinates. Generalized forces for objects are summed on the so-called right hand side (RHS) for according nodal coordinates, independently whether they represent applied forces of loads or internal (e.g., elastic) forces of elastic bodies. Solvers then make use of these system functions in order to compute residuals. Furthermore, Jacobians can be computed by the system (currently not in parallel). The system Jacobian is assembled from object Jacobians, using their internal mode of computation, which can be either numerical, analytical or by means of automatic differentiation. For verification, some switches exist to compare analytic Jacobians against their numerical counterpart.

4.2.3 Coordinates and local-to-global mapping

The difference between (local) computational objects and the (global) system is the mapping of local (unknown) coordinates to global coordinates. This mapping is denoted as local to global (LTG) mapping, which is essential both for the computation of object quantities, such as the position or deformation, as well as for the action of forces onto objects. In general, the LTG-mapping follows the node numbering, where system coordinates $\mathbf{q}$ consist of node i coordinates $\mathbf{q}_{nODE2,i}$,

$$\mathbf{q} = [\mathbf{q}_{nODE2,0}^{\mathrm{T}}, \mathbf{q}_{nODE2,1}^{\mathrm{T}}, \dots] \tag{1}$$

The system allows to include first and second order differential equations (ODE) together with algebraic equations, with according coordinates. The set of available coordinate types is:

- coordinates for first order differential equations (ODE1)
- coordinates for second order differential equations (ODE2)
- coordinates for algebraic equations (AE)
- data (history) coordinates (Data)

Eq. (1) holds for any of the above mentioned coordinate types and assembles the four main system coordinate vectors $\mathbf{q}$ for ODE2, $\mathbf{y}$ for ODE1, λ for AE, $\mathbf{x}$ for Data. While it would be sufficient to have ODE1 coordinates instead of ODE2 coordinates, it can boost efficiency and simplify implementation using ODE2 coordinates with an associated mass matrix and known relations between position, velocity and acceleration. In addition to algebraic coordinates, there is also need for data coordinates or history variables, which can store information on previous steps, not related to physical quantities, such as plastic strains, slip or other contact variables.

4.3 Functionality of the present code

The current Exudyn version 1.3.75.dev1 offers a considerable capability to solve rigid and flexible multibody systems. In particular there exist standard rigid bodies and point masses, both in 2D as well as in 3D. As there are many applications which are fully planar, special functionality exists for 2D bodies – in particular for performance reasons, while the general formalism is always 3D. Additionally, 1D objects can be used for simple mechanisms or in order to create linear or rotational motion just w.r.t. one fixed axis.
Flexible bodies can be created using the floating frame of reference formulation. This formulation offers a reduced order version, which can incorporate any modal reduction. Import of mesh and modal reduction can be done with the included FEM module, which allows to realize standard mechanisms straightforward. Furthermore, external finite element tools, such as NGsolve [14], are attached to enable direct coupling with large-deformation solid (tetrahedral) finite elements.
Large deformation cable elements exist in 2D, also including special techniques for axially moving beams, while 3D rods (beam) are currently under development and possess restricted functionality.
In most cases, researchers will not find one of their required components, such as special flexible bodies or connectors. For this reason, there exist generic objects, such as `ObjectGenericODE2` or `ObjectGenericODE1`, which allow to create any mechanical system based on differential equations defined in Python. Furthermore, connectors can be extended or modified by means of Python user functions, e.g., in order to create special force laws, or for loads to obtain arbitrary time-dependent values. Python user functions have the advantage that any change does not require re-compilation and user functions can make usage of (nearly) every Python package including scientific computing (e.g., `scipy`) or artificial intelligence (e.g., `tensorflow`). While Python user functions may be a factor of 100 to 1000 times slower than their C++ correspondent[7], it may be a convenient alternative to modify the underlying C++ code. Besides the inevitable overhead for Python function calls, pure Python and `numpy` functions can make use of just-in-time compilation, e.g., using the Python tool `numba` [8], which reduces the Python overhead solely to the interfacing between Exudyn and Python.

4.4 Solvers

In order to optimize performance, the implementation of system-specific solvers is inevitable. For the possibility to perform different analysis of multibody systems, several solvers are essential:

- Explicit dynamic solver: for efficient solution of non-stiff problems with solely ordinary differential equations, e.g., rigid bodies under contact or kinematic trees
- Implicit dynamic solver: for solution of stiff or differential-algebraic equations
- Static solver: in order to perform kinematic analysis or for computation of initial values for static equilibrium

Further solvers can make use of the dynamic and static solvers, e.g., optimization or sensitivity analysis, shooting methods or numerical differentiation. Furthermore, the Jacobian of the dynamic implicit solver is used to obtain the linearization at a specific configuration, allowing to perform an eigenvalue analysis for this linearized system. Certainly, some advanced solvers are not implemented, such as an adjoint solver for the dynamic solution.
Computational expenses for the computation of a single static or dynamic step are distributed across several computational functions of the multibody system. These functions mostly represent the residual of the equations of motion

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \frac{\partial \mathbf{g}}{\partial \mathbf{q}^{\mathrm{T}}}\lambda = \mathbf{f}(\mathbf{q},\dot{\mathbf{q}},t) \tag{2}$$

$$\dot{\mathbf{y}} + \frac{\partial \mathbf{g}}{\partial \mathbf{y}^{\mathrm{T}}}\lambda = \mathbf{f}_{ODE1}(\mathbf{y},t) \tag{3}$$

$$\mathbf{g}(\mathbf{q},\dot{\mathbf{q}},\mathbf{y},\lambda,t) = 0 \tag{4}$$

[7] The pure call to Python functions via pybind11 leads to an overhead of approximately 1 μs, while C++ can call item functions in less than 10 ns. Costs increase linearly, if Exudyn interface functions are called within user functions.

System level computational functions are split into computation of the state-dependent mass matrix $\mathbf{M}(\mathbf{q})$, the right-hand-side $\mathbf{f}(\mathbf{q},\dot{\mathbf{q}},t)$ of ODE2 equations, the right-hand-side $\mathbf{f}_{ODE1}(\mathbf{q},\dot{\mathbf{q}},t)$ of ODE1 equations, the residual of constraint euqations $\mathbf{g}(\mathbf{q},\dot{\mathbf{q}},\mathbf{y},\lambda,t)$, reaction forces $\frac{\partial \mathbf{g}}{\partial \mathbf{q}^T}\lambda$ and Jacobians of Eq. (2) w.r.t. the according static or dynamic solver. Note that the linear system of equations in the Newton method is solved either by a simplistic dense solver with few overhead and no memory allocation for small scale systems, or by the Eigen[8] based `SparseLU` solver, using a super nodal approach for factorization of non-symmetric sparse matrices. It should be mentioned that Eigen is only used for sparse systems, while dense linear algebra is fully implemented in Exudyn, as this leads to much shorter compilation times and simpler debugging. For large system vector operations, both multithreaded parallelization as well as Intel's advanced vector extensions (AVX) are utilized.
In order to avoid re-parameterization of rotation parameters in explicit dynamic simulations, a Lie group formulation has been developed for Exudyn which is able to directly update rotation parameters using the rotation vector [6]. Solvers and rigid bodies (and in future beam elements) are adapted to special Lie group nodes, which simplify kinematic operations and may lead to a constant mass matrix.
Specifically for large scale models as well as for cases where many simulation runs are required (e.g., parameter variation or optimization), short CPU times are needed. Due to the large variety of problems, ranging from 1-DOF to 1 000 000 DOF problems, tuning of solvers is required and usually may influence the CPU times by a remarkable factor. Specifically, we mention switches between dense and sparse matrix computation, avoiding Python calls within user functions, adjust Newton and discontinuous iteration parameters, e.g., switch to modified Newton. Whenever possible, explicit solvers should be used. Systems with purely constant mass matrices (e.g., using Lie group nodes or ANCF elements) avoid the time-consuming rebuilding of the matrix. In order to determine the time consuming parts of the solver, an integrated facility runs various timers and allows to show the amount of time spent, e.g., in Jacobian computation, file writing or visualization, which allows the user to adjust the solver settings for performance improvements. Whenever repeated simulations of a parameterized model are required, a Python utility function `ParameterVariation` exists, which allows to perform multithreaded runs of this variations with excellent scalability including a mode to run on a cluster. For detailed description also see the section 'Performance and ways to speed up' in theDoc.pdf [4].

4.5 Multithreaded parallelization

Finally, Exudyn has been recently extended for multithreaded parallelization, which includes parallelized computation of mass matrix, right-hand-sides (elastic forces and loads), reaction forces and of all functions in the contact module. There exist two different task managers which are compared in this paper. The preferred version is taken from NGsolve [14], which includes a simple thread-based parallelization, which starts up a number of given threads, which continuously wait for new tasks to work with. A simple load management is used to handle tasks (such as the computation of a body's mass matrix) with different CPU times. A modified version, with a minimum inter-task communication and no load management (therefore suited for tasks with equal loading only), has been derived and denoted as tiny task manager. As will be seen in the examples, the tiny task manager is able to effectively parallelize very small systems. Note that most overhead for the task manager is due to the inevitable synchronization of atomic variables for all used tasks at CPU-level, needed at the beginning and end of a set of tasks and for load management. This synchronization time can be much larger than the computation time of the task itself, see the results of the mass-spring-damper system. Note that both task managers are tested within two different compilations of Exudyn, in which NGsolve[14] is the default one, also available with pip installers.
Parallelization may be applied to any for loop at any computationally relevant function of the code, reading in the serial version (in pseudo C++ code):

```
for (int i=0; i < numberOfBodies; i++)
{
  const RigidBody& rigid = system.GetBody(i);
  const Vector& LTG = system.GetLTG(i);
  rigid.ComputeResidual(objectRHS);
  systemRHS.AddVector(objectRHS, LTG);
}
```

[8] https://eigen.tuxfamily.org

This function computes the object's RHS and adds this vector to the corresponding coordinates, defined by the LTG mapping in the system RHS. The parallelized version is realized via C++ lambda functions and called `ParallelFor(...)` [14]:

```
ParallelFor(numberOfBodies, [&system, &systemRHS, &objectRHS, &numberOfBodies](int i)
{
  const RigidBody& rigid = system.GetBody(i);
  const Vector& LTG = system.GetLTG(i);
  rigid.ComputeResidual(objectRHS);
  temp[GetThreadID()].AddVector(objectRHS, LTG);
})
systemRHS.AddVectors(temp);
```

In the parallel version, the for loop runs independently for the number of threads defined during initialization. Therefore, data is written into thread-local vectors temp[GetThreadID()], which are finally added to the system RHS. The final step is done in serial mode, because currently solvers are only available in serial mode, as there exists no sparse solver with according interface.
Finally, there is a special module for contact computations, allowing to efficiently compute contact between spherical particles (or clusters), triangulated surfaces and beams. This module is decoupled from the otherwise item-based structure of the code and thus allows highly optimized computations using search trees and advanced parallelization techniques.

5 Advanced features

This section highlights some unique features, that are designed to help the user to conveniently create multibody models.

5.1 Python interface, user functions, naming conventions and parameterized models

Models can be created not only by the Python interface, but also by using many convenient functions of Python modules which are integrated on the Python side of Exudyn. These modules allow to create parameterized models, simply add rigid bodies with inertia transformation, joint-axis computation or add graphics. A finite element preprocessing tool allows to import data from ABAQUS or Ansys or can use the deeply integrated NGsolve package [13, 14] from `https://ngsolve.org`. Well known Python packages such as scipy are integrated to compute eigenmodes, system linearization, optimization or sensitivity analysis. After adjusting any of the 400 simulation and visualization settings (or leaving defaults), users need to assemble and start solvers to create complex simulation of real life systems.
In many situations in research and industrial applications there are features missing in order to perform the desired multibody dynamics simulations. While one may circumvent the missing features with workarounds, it is convenient to use a large set of Python user functions and system callbacks, which enable to create user-defined computational objects (such as bodies, connectors or joints) as well as user-defined solvers or system behavior, e.g., in order to model breaking mechanisms or similar.
In the early days of scientific computing and finite elements, computer languages such as FORTRAN limited the number of characters per line and programmers were using short variable names (`i`, `j`, `n`) and function names such as LAPACK's `dgetrf` for LU factorization. Exudyn tries to follow modern coding standards, which abandon short names in favor of code readability, using comments for helpful information rather than translation of variable and function names. Due to type completion, there is no need to use short names, as the user can select from a list of classes or functions when operating with certain libraries or data types. Using full names, such as `massMatrix`, immediately leads to readable scripts, reduces the amount of errors due to mistyping or double use of variable names and leads to faster coding, as there is always one unique (non-abbreviated) name for every function or variable. Furthermore, Exudyn uses the camel case convention throughout, in which variables start lower case and function or classes start with upper case characters.
Finally, parameterized models follow naturally, which either may varied manually, or automatically by means of advanced Exudyn Python functions, such as `ParameterVariation(...)`. Parameterized models can be directly optimized with `GeneticOptimization(...)` or `Minimize(...)`.

5.2 Automatic code generation

As part of the design, code duplication is avoided as much as possible. This mostly affects data structures and interface functions. The more the data structures are open to the user, e.g., being able to adjust some special solver settings, and the more interface functions exist, e.g., in order retrieve some current information on the state of the system or solver, the easier and safer the usage gets. Extending interfaces may be time consuming and programmers may be reluctant to extend interfaces, because they always need specification of the interface structure, some documentation (e.g., about special behaviour), definition and checking of valid ranges, and interfaces need to create code at several places. As for the scalar mass of a rigid body, we have the read and write functions, the data variable for mass, the Python interface function (C++ and Python), the range checking function and the documentation. This gives at least seven places at which code needs to be created (and possibly documented). In the current implementation, for interfacing only, a regular object data variable is repeated 21 times in the C++ and Python code. All of these code parts are auto-created from a single definition, which allows to change or adapt such interfaces easily. It also avoids that some of these interfacing functions are missing or wrong and untested.

5.3 User friendlyness

One of the most annoying features of open source codes is often that it is not known what happens behind the scenes. It is therefore an integrated feature, as already available in Hotint [2], to immediately show the current simulation state in a 3D visualization. In this way, main kinematic errors can be detected instantaneously. The highly integrated graphics renderer is based on the multi-platform, but tiny tool GLFW[9]. This is convenient for interactive models in teaching but also lets students create models within short time.

Special emphasis has been laid on pre-checking, error messages and warning, which are an integral part of Exudyn. Due to the fact that C++ errors are usually unreadable for the user, pre-checks are performed, which usually show approximately the reason and place of the error. Furthermore, a large number of range and validity checks are performed in the regular version of Exudyn, which leads to a significant, typically 30% overhead. Therefore, specially compiled C++ sub-modules exist, denoted as exudynCPP-fast, which do not include range checks and timing measurements, thus giving full performance. As a further example, Python user functions are called from the C++ module, which would usually lead to cumbersome internal error messages ('Exception raised in solver'). Therefore, Exudyn has special exception barriers around all Python user functions, allowing to return information on place and type of the error.

In addition to common Python packages, such as `numpy`, `sympy` or `scipy`, specific Exudyn Python utilities have been added in order to conveniently create multibody models. These utilities include the aforementioned finite element preprocessing tools, but also basic functions for rigid bodies (e.g., Euler parameter helper routines, rotation matrix computations, homogeneous transformations, and Lie group utilities), robotics, artificial intelligence, graphics import and creation, as well as an advanced interface to `matplotlib`, which allows to display sensor results in a one-liner. An interactive module allows to create models which are continuously running with possible interaction with the user, e.g., to demonstrate the behavior of a nonlinear oscillator or a gyro while changing system parameters or loads.

The current documentation, which is closely linked to the interface definition of every item and automatically created, has more than 700 pages, see the pdf document `docs/theDoc/theDoc.pdf` on github [4]. Furthermore, there are tutorials and demos available on Youtube [3].

6 Create simple example in Exudyn

This section is intended to provide a simple but fully functional example in Python. Requirements are an installed Miniconda with `matplotlib` installed or a full Anaconda (recommended). The example runs on a wide range of Exudyn and Python versions (3.6 – 3.10), but has been tested with Exudyn V1.3.74.dev1. The example models a rigid pendulum, using a redundant coordinate formulation based on a rigid body (using Euler parameters) and a generic joint, restricting all relative translations and rotations except for rotation around the local z-axis:

Most of the example script shall be self-explanatory based on given comments. The example follows the standard Exudyn script structure, as follows:

[9] `https://www.glfw.org`

1. import exudyn, exudyn.utilities and other required packages
2. create a system container and a mbs system (but: there can be more than one container or system)
3. add bodies to mbs (in more general case, first nodes are added, then objects)
4. add joints to mbs
5. add loads and sensors to mbs
6. assemble mbs
7. define simulation and visualization settings
8. start simulation of mbs

Optionally, as shown in the following example script, the 3D renderer may be started before simulation, in order to show the updated state of the system during simulation.

```
#import exudyn modules and numpy:
import exudyn as exu
from exudyn.utilities import * #also imports itemInterface and graphics
from exudyn.plot import PlotSensor

#create empty multibody system
SC = exu.SystemContainer()
mbs = SC.AddSystem()

#parameters:
g = [0,-9.81,0] #gravity
L = 1           #body dimensions
b = 0.1         #body dimensions

#create inertia and body graphics
iCube = InertiaCuboid(density=5000, sideLengths=[L,b,b])
graphics=GraphicsDataRigidLink(p0=[-0.5*L,0,0],p1=[0.5*L,0,0],axis0=[0,0,1],
           radius=[0.5*b,0.5*b], thickness=b, width=[b,b], color=color4red)
#add rigid body (in background creates node and object)
[n0,b0]=AddRigidBody(mainSys = mbs, inertia = iCube,
                     nodeType = exu.NodeType.RotationEulerParameters,
                     position = [L*0.5,0,0], gravity = g,
                     graphicsDataList = [graphics])
#ground body and markers
oGround = mbs.AddObject(ObjectGround())
markerGround=mbs.AddMarker(MarkerBodyRigid(bodyNumber=oGround,
                                localPosition=[0,0,0]))
markerBody0J0 = mbs.AddMarker(MarkerBodyRigid(bodyNumber=b0,
                                   localPosition=[-0.5*L,0,0]))
#revolute joint (free z-axis)
mbs.AddObject(GenericJoint(markerNumbers=[markerGround, markerBody0J0],
            constrainedAxes=[1,1,1,1,1,0],
            visualization=VObjectJointGeneric(axesRadius=0.01, axesLength=0.1)))

sPos = mbs.AddSensor(SensorNode(nodeNumber=n0, fileName='pos.txt',
                        outputVariableType=exu.OutputVariableType.Position))

#finalize mbs, prepare for visualization and simulation:
mbs.Assemble()

#settings (out of several 100 options, using defaults):
simulationSettings = exu.SimulationSettings()
simulationSettings.timeIntegration.numberOfSteps = 1000
simulationSettings.timeIntegration.endTime = 4

#start visualization and solver:
exu.StartRenderer()
mbs.WaitForUserToContinue()
exu.SolveDynamic(mbs, simulationSettings)
exu.StopRenderer() #safely close rendering window!

#plot result of sensor 'sPos'
PlotSensor(mbs, sPos, closeAll=True)
```

The graphical output of this example can be seen in Fig. 4. When pressing space, simulation finishes in less than 0.05 seconds and the output of the sensor is shown. For more detailed examples, see [4].

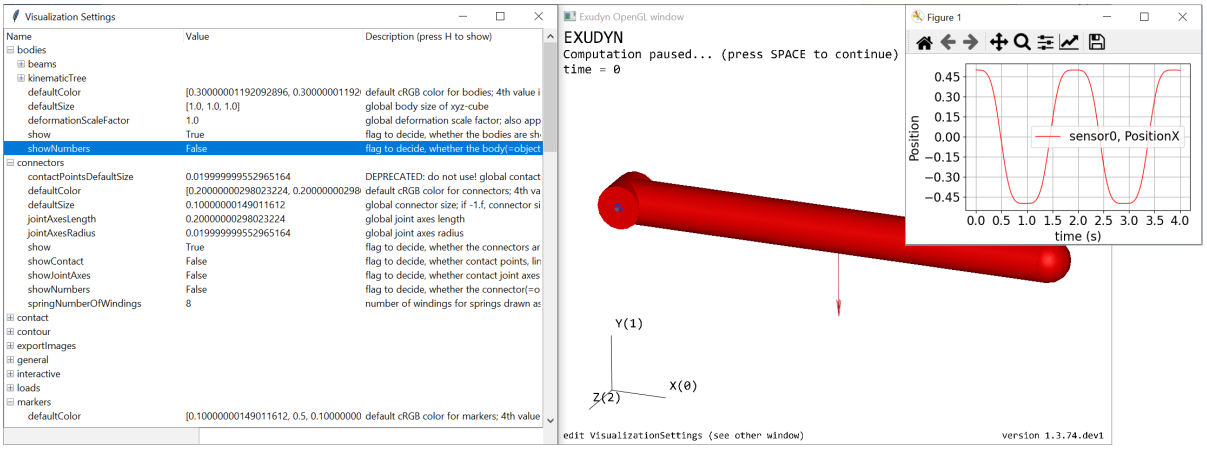

Figure 4: Graphical representation of simple rigid body example (middle), visualization settings (left) and `PlotSensor(...)` output (right)

7 Performance tests and evaluation

The present section evaluates 5 different small to medium sized examples in order to demonstrate performance. Specifically, the benefits of multithreaded parallelization are demonstrated for a small as well as moderately large number of multibody components or coordinates. The tests are performed on a desktop workstation PC with one Intel Core i9-7940X CPU with 14 cores (supporting 28 threads via hyper-threading), 3.10GHz, 96 GB RAM, Windows 10 which is denoted as 'Intel i9 (14 cores) test computer ' in the following. Note that regular runs on other CPU architectures show similar behavior as reported here, including mobile, AMD, and Apple M1 processors, while performance on Linux is better, in general. As it is well known that hyper-threading does not improve performance if the underlying code is optimized and using all floating point units or reaching memory bandwidth limits, the performance is only evaluated up to 14 parallel threads.

The overall performance of the code is not specifically compared with other codes. However, to give an idea, the CPU time of the stiff flyball governor[10] of the IFToMM multibody benchmarks is used and compared for a similar implicit trapezoidal integration method and fixed step size of $5 \cdot 10^{-4}$ s. The comparison on comparable platforms and CPUS, using only serial computation, shows:

- IFToMM reference of MbsLab: Intel Core i7-4790K@4.00 GHz, 8 GB RAM, Windows 8.1 Pro, 64 bit: 0.853 s CPU time
- Exudyn / minimum coordinates: Intel i9 (14 cores) test computer : 0.67 s CPU time
- Exudyn / redundant coordinates: Intel i9 (14 cores) test computer : 1.39 s CPU time

This comparison shows the good overall performance of Exudyn, being a general rigid-flexible code, but also demonstrates the large differences of two diverse multibody formalisms. The fully redundant formulation leads to 28 nonlinear ODE2 equations and 28 nonlinear algebraic constraints, which are solved within approximately 75 000 iterations of a modified Newton scheme, where most time is spent for computation of constraint reaction forces and solving the linear equations in Newton. In the minimum coordinates (kinematic tree) case, the system has only four equations of motion, which are highly coupled. Most of the time is spent for computation of the spring-damper forces as they require computation of the full tree kinematics for each connection point – with potential for optimization in the future.

Thus, further tests focus on the performance using multithreading parallelization, which is nowadays available on every workstation, laptop, but also low cost computer. In all tests, the solver is using sparse matrices, except for the mass-spring-damper system with less than 20 unknowns, which switches to dense matrices in this case. For implicit time integration, the modified Newton method is used, which only updates Jacobians in case of slow convergence. Furthermore, visualization as well as file output is turned off and the computer is running almost idle.

[10] `https://www.iftomm-multibody.org/benchmark/problem/Stiff_flyball_governor`

7.1 Mass-spring-damper system

The first example is a mass-spring-damper system. The system consists of n mass points, modeled with 1D nodes (having only one unknown), and a `CoordinateSpringDamper`, which acts on the coordinates of two adjacent nodes. There is only one scalar, linear equation of motion for every mass and spring damper, such that most of the computational efforts are overheads created within the system calls. We simulate 100 000 time steps over 100 seconds of time, which results in less than 0.1 seconds of computation time for the single-threaded one-mass system using (implicit) generalized alpha method. Due to speed measurements, some overheads add up, which otherwise would lead to considerably less than 0.1 seconds of simulation time.

In this example, two multithreading approaches (NGsolve and tiny task manager) are compared. The general overhead for multithreading, independently of the number of masses, is measured. For the tiny task manager, the general overhead is almost independent of the number of threads and measured as $\approx 3.5\,\mu$s per step. For the original NGsolve taskmanger, the general overhead is $\approx 1\,\mu$s per thread and per step. However, due to load balancing, additional overheads occur in the NGsolve task manager, measured as $\approx 4\,\mu$s per step. For larger number of masses, load balancing helps to result in better results for the NGsolve task manager (therefore it is used by default). Note that these times are valid for the Intel i9 (14 cores) test computer and may be different for other hardware.

The CPU times for different number of masses, threads and task managers are shown in Fig. 5 and 6. Remarkably, in case of 1 thread, which fully turns off multithreading routines, the CPU results differ significantly for the two multithreading approaches. Nevertheless, for 10 000 masses, the speedup is 4.9 for the tiny threading approach and 4.6 for original NGsolve. Furthermore, the break-even for 4 threads is at 25 masses in the tiny task manager and at 40 masses for the NGsolve version. Even worse, the break-even for 12 threads is at 30 masses in the tiny task manager and at 70 masses for the NGsolve version.

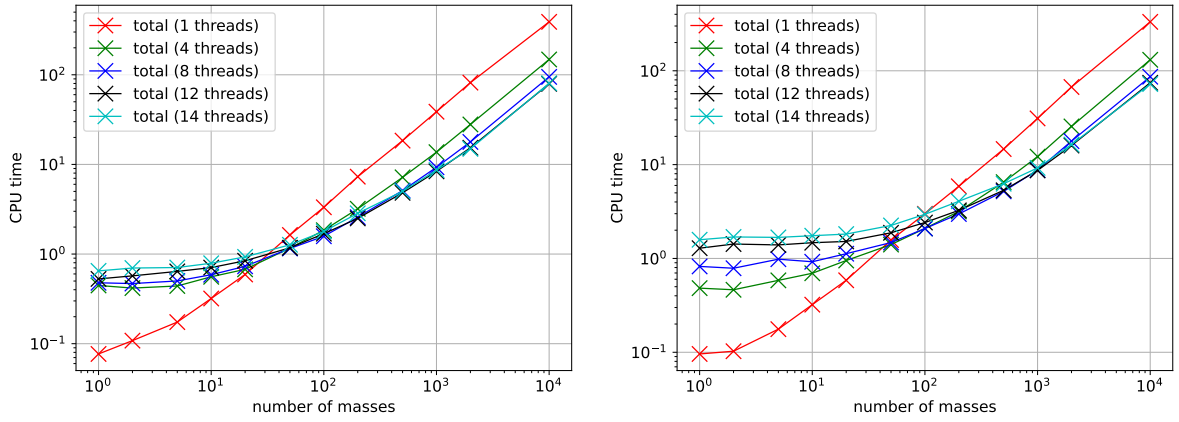

Figure 5: CPU times versus number of masses in mass-spring-damper system for different multithreading approaches on Intel i9 (14 cores) test computer using a logarithmic scale; Tiny task manager (left) and NGsolve original threading (right)

7.2 Robot: redundant coordinates

A more relevant multibody system is based on a test with n_m serial manipulators with 6 DOF each, see Fig. 7. In the first test set, redundant coordinates and constraints are used. This gives $6 \times n_m$ rigid bodies (using 4 Euler parameters for rotations), and $6 \times n_m$ constraints, leading to $42 \times n_m$ ODE2 coordinates per robot as well as $42 \times n_m$ constraints. Due to a generic joint approach, $6 \times n_m$ constraints act on the Lagrange multipliers for the free rotation axis, being always zero. Furthermore, the drives of the robots are simple PD-controllers realized as spring-dampers with prescribed motion. Note that all robots move on different trajectories in order to avoid an overly simple structure of the problem. The implicit generalized alpha solver is used to solve the system for 1 s with 1000 steps.

In this test case, the number of robots is varied and CPU times are measured for different number of threads, see Fig. 8 and 9. Both cases lead to nearly identical parallelization effects, while the tiny task manager performs better for few robots and large number of treads. Note that in case of micro-threading

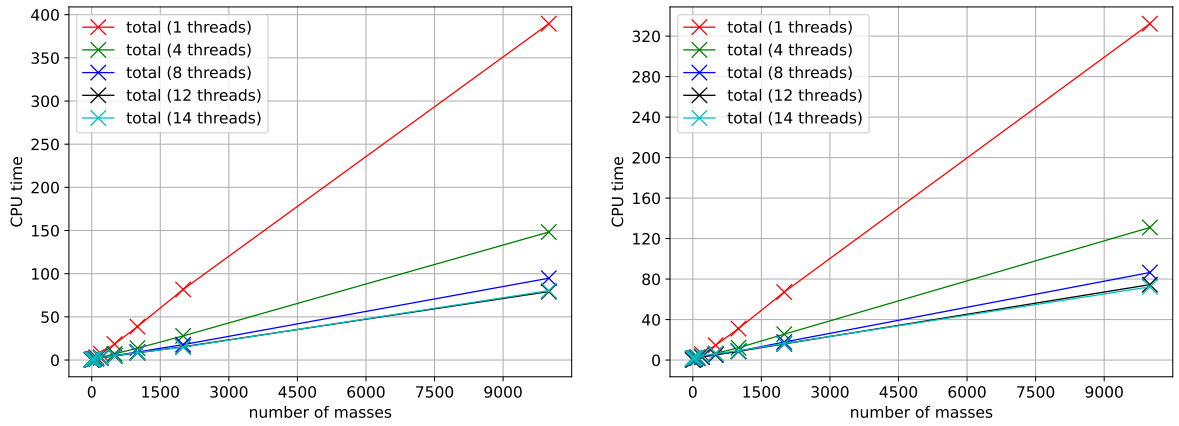

Figure 6: CPU times versus number of masses in mass-spring-damper system for different multithreading approaches on Intel i9 (14 cores) test computer ; Tiny task manager (left) and NGsolve original threading (right)

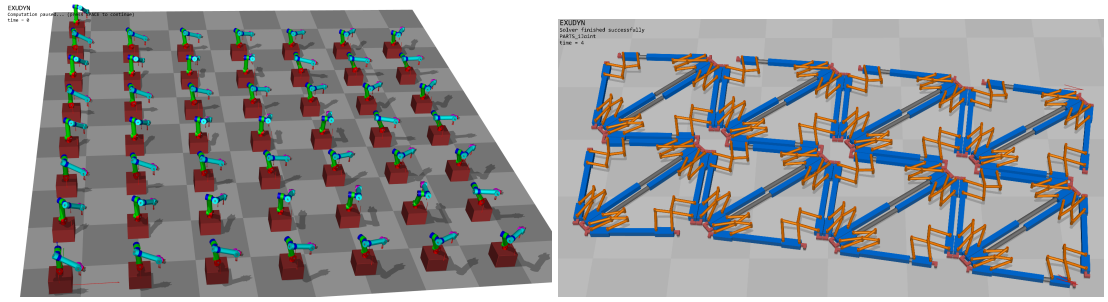

Figure 7: Left: test case with serial manipulators; the number of manipulators is varied in this test, while 50 robots are shown exemplary; right: test case with 4×2 cellular robots

already one robot sees little parallelization speedup with only 6 bodies involved. The speedup for the NGsolve task manager is 2.5 for larger number of robots. Note that for 500 robots the system has 21 000 unknowns to be solved. Thus, the Jacobian computation and the solver cause 25% of computation time in the serial version, causing a lower speedup factor.

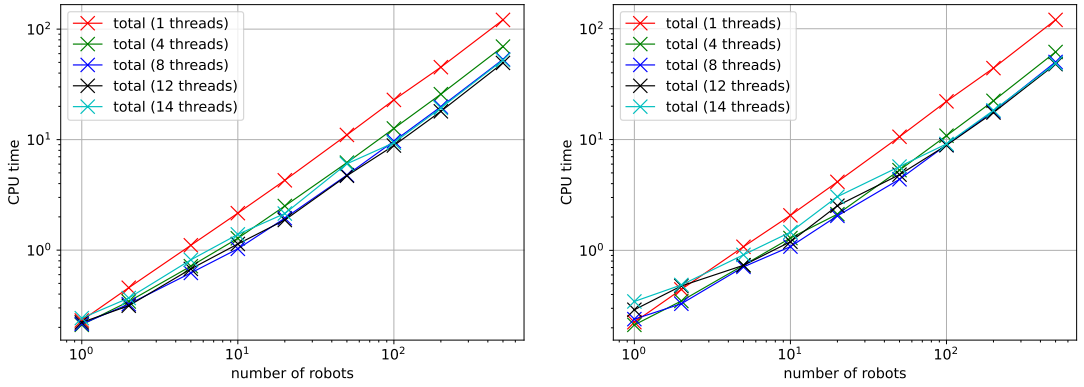

Figure 8: CPU times versus number of robots with redundant coordinates for different multithreading approaches on Intel i9 (14 cores) test computer using a logarithmic scale; Tiny task manager (left) and NGsolve original threading (right)

7.3 Robot: minimum coordinates formulation

In a second typical multibody example, the system under investigation is the same as in the previous test, however, using a minimum coordinates formulation. This leads to only $6 \times n_m$ unknowns in the system and no constraints. Furthermore, an explicit 4th order Runge Kutta time integrator is used to solve the system. The system is again solved for 1 s with 1000 steps.

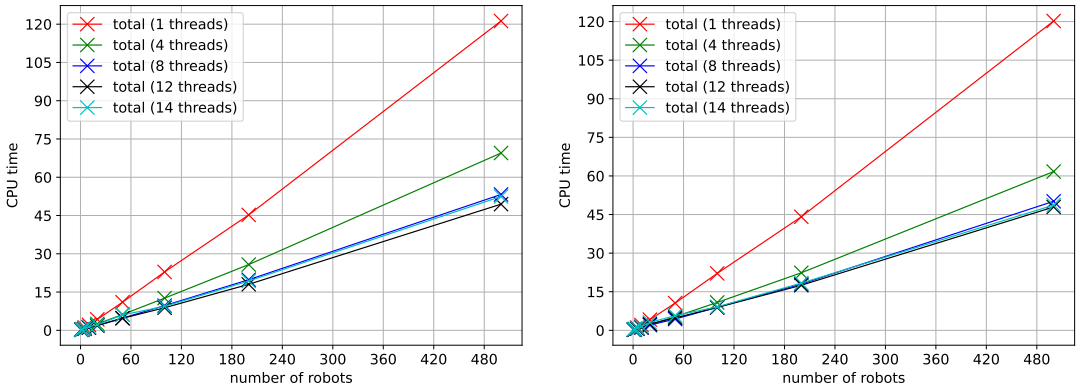

Figure 9: CPU times versus number of robots with redundant coordinates for different multithreading approaches on Intel i9 (14 cores) test computer ; Tiny task manager (left) and NGsolve original threading (right)

In this test case, the number of robots is varied and CPU times are measured for different number of threads, see Fig. 10 and 11. Both cases lead to nearly identical parallelization effects, while the tiny task manager performs better for small number of robots and large number of threads, see Fig. 10. The speedup for the NGsolve task manager is 2.5 for larger number of robots, showing limitations due to serial factorization of the mass matrix in the explicit integrator.

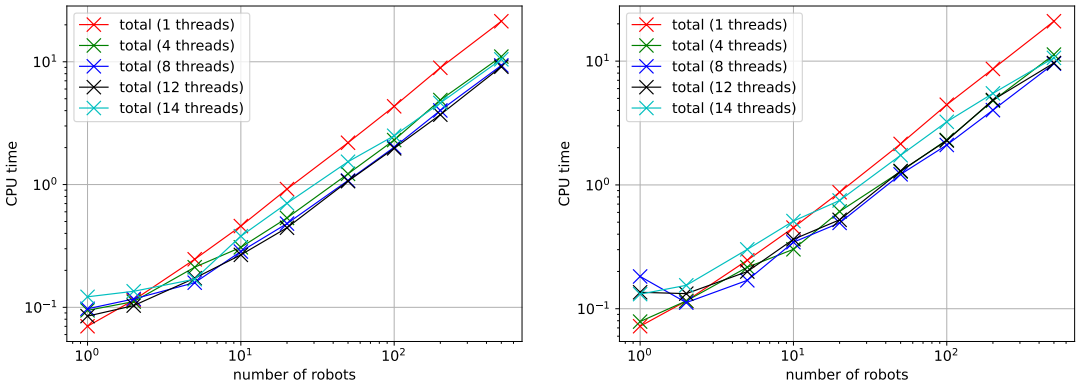

Figure 10: CPU times versus number of robots with minimum coordinates for different multithreading approaches on Intel i9 (14 cores) test computer using a logarithmic scale; Tiny task manager (left) and NGsolve original threading (right)

7.4 Cellular robot PARTS

The final rigid body example contains a highly parallel mechanism modeling cellular robots known as PARTS [12]. The system consists of triangular cells with actuators at each triangle side and special six-bar-linkages at the vertices.
The system with 16 triangles as shown in Fig. 7 consists of 336 (planar) revolute joints, 288 planar rigid bodies and 1641 total unknowns to be solved in every step of the implicit generalized alpha integrator. A second system of double size with 32 triangles and 576 rigid bodies is investigated, as well. The CPU times for both systems are shown in Tab. 1, leading to a maximum speedup factor of 3.4 for the larger system.

7.5 Belt drive

The first flexible multibody system represents a belt drive similar to the problem described in [11]. The belt drive consists of two pulleys under contact with a belt discretized with 480 ANCF elements following the formulation of [5]. The total degrees of freedom including the 2D rigid bodies of the sheaves is 1926. The system is solved for 0.1 s with 1000 total time steps and a constant step size. In the serial version (1 thread), most computational time of the implicit trapezoidal rule results in the evaluation of the elastic

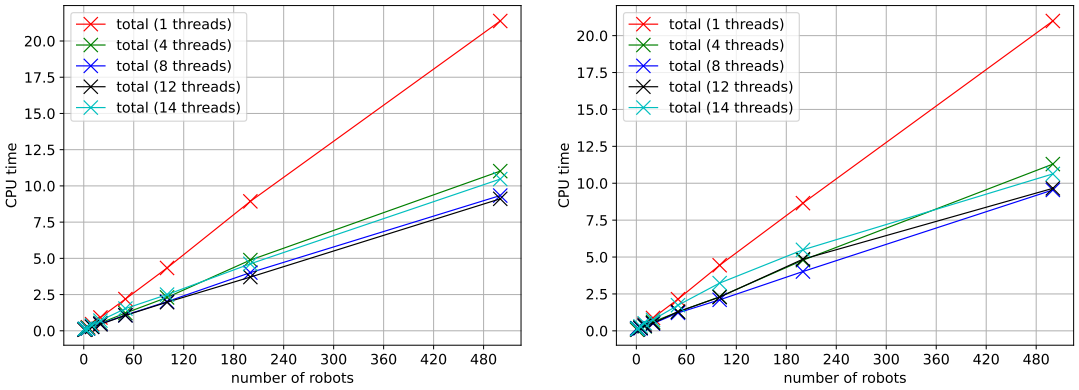

Figure 11: CPU times versus number of robots with minimum coordinates for different multithreading approaches on Intel i9 (14 cores) test computer ; Tiny task manager (left) and NGsolve original threading (right)

Table 1: CPU times for simulation of PARTS for 20 s of simulation time, Python 3.7, Exudyn 1.3.50 without range checks, NGsolve task manager; Intel i9 (14 cores) test computer

number of threads	rigid bodies	time steps	1	4	8	10	12
CPU time (seconds)	288	2000	8.42	3.31	3.01	3.06	3.76
CPU time (seconds)	576	1000	8.06	3.15	2.51	2.34	2.42

forces and the contact law between beam elements and pulleys (77.4%), nonlinear iterations related to contact (10.9%), and the back-substitution for the Newton increments (9.8%). Due to the fact that the mass matrix is constant, its evaluation time is negligible. The small time step leads to only 6 updates of the Jacobian with 5635 iterations in total for the modified Newton method. Note that a simulation with full Newton method would result in only 2238 Newton iterations, but resulting in 102 s computation time, 70.5% for the Jacobian evaluation and 20.9% for factorization and back-substitution. Even if the Jacobian evaluation could be parallelized, factorization would itself require more CPU time than the computations with modified Newton method.
The speedup due to multithreaded computation is shown in Tab. 2, showing a maximum speedup factor of 3.31. Note that the limitation is due to the increasing amount of serial solver time, while the evaluation of elastic forces shows speedups of 7. As can be observed in Tab. 2, the optimal value is located around 10-12 parallel threads of 28 available (via hyper-threading).

7.6 Piston engine

A comparatively large model is tested based on a deformable 18-piston engine, see figure Fig. 12. All 37 bodies are flexible bodies using quadratic tetrahedral elements and the floating frame of reference formulation as published in [16]. The bodies have a total of 368391 original DOF. The Python package NGsolve [14] is used to mesh the flexible bodies, which also allows to export mass and stiffness matrices for each body.
The original DOF are reduced with the help of Exudyn's `FEM` module for every body using free-free eigenmodes, even though that other modal reduction methods may be more accurate. The total time for preprocessing including geometry creation, meshing, mode computation and computation of stress

Table 2: CPU times for belt drive with implicit time integration, Python 3.8, Exudyn 1.3.74 without range checks; Intel i9 (14 cores) test computer

number of threads	1	2	4	6	8	10	12	14	20	28
CPU time (seconds)	11.5	7.62	4.95	3.97	3.71	3.47	3.49	3.84	6.04	6.87

Table 3: CPU times for piston engine with implicit time integration, Python 3.10, Exudyn 1.3.70, NGsolve multithreading, Intel i9 (14 cores) test computer

number of threads	1	4	12
CPU time (seconds)	15.5	6.36	4.20

modes only takes 72 s CPU time, whereof eigenmode computation within a special Exudyn-NGsolve Python functionality takes 38.8 s seconds. This easily allows to perform parameter variations for the complete flexible multibody system.
The simulation is performed for 0.01 s with 400 constant time steps, resulting in the CPU times given in Tab. 3 with a total speedup factor of 3.7 for 12 cores. This example again demonstrates the high potential of parallelization even in the case of a small number of bodies.

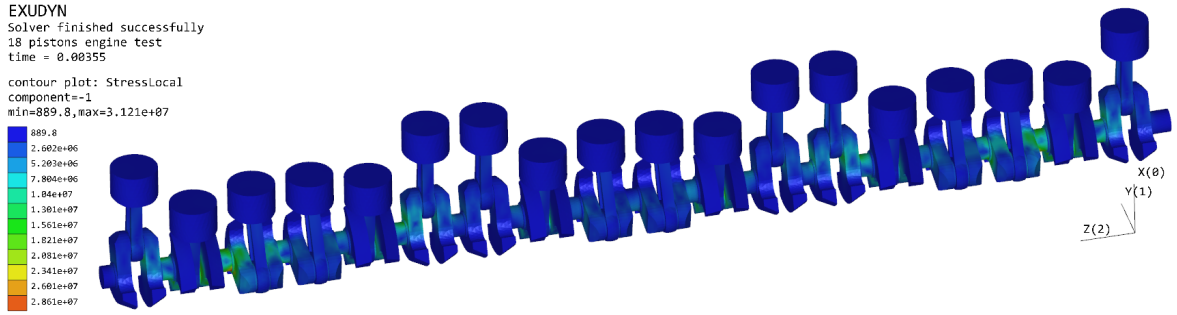

Figure 12: Test model of a 18-piston engine with 37 flexible bodies

Conclusions and outlook

A new general purpose C++ flexible multibody dynamics code has been developed and put open source. The paper introduces the main architecture and features of the code. In particular computational performance is evaluated for multithreaded parallelization using novel threading approaches adapted for very short tasks. Tests reveal that parallelization is effective for as few as 100 degrees of freedom in very simple mass-spring-damper systems, with a maximum speedup factor for larger systems close to 5 for a 14 cores test computer. Furthermore, in more advanced rigid and flexible multibody systems, parallelization significantly reduces CPU time with as little as 12 rigid bodies.
Future developments will focus on parallelized solvers, as factorization and back-substitution currently cannot run in parallel with the available Eigen SparseLU solver. While the code has made significant progress since its foundation in 2019, there is still need for large deformation 3D beam elements, improved computation of Jacobians and an improved contact solver, which will be subject of future research and development.

Acknowledgments and thanks

The project could not have reached the current state without significant help from many colleagues. The team at the Department of Mechatronics is gratefully acknowledged for supporting with computer infrastructure and testing. In particular, contributions to the ConvexRoll (Peter Manzl), nonlinear beams (Michael Pieber), robotics (Martin Sereinig) and the Lie group formulation (Stefan Holzinger).
Joachim Schöberl (TU Vienna) and his group contributed significantly to the parallelization and the coupling to his finite element code NGsolve. Andreas Zwölfer (TU Munich) contributed to the development of the floating frame of reference formulation, which is highly simplified as compared to standard integral-based formulations. Further acknowledgments are mentioned at github. Thanks to all!

References

[1] E. Coumans. Bullet physics simulation. In *ACM SIGGRAPH 2015 Courses*, SIGGRAPH '15, New York, NY, USA, 2015. Association for Computing Machinery.

[2] J. Gerstmayr. HOTINT – A C++ Environment for the simulation of multibody dynamics systems and finite elements. In K. Arczewski, J. Fraczek, and M. Wojtyra, editors, *Proceedings of the Multibody Dynamics 2009 Eccomas Thematic Conference*, 2009.

[3] J. Gerstmayr. Exudyn – Youtube channel. https://www.youtube.com/playlist?list=PLZduTa9mdcmOh5KVUqatD9GzVg_jtl6fx (accessed on April 29, 2022), 2022.

[4] J. Gerstmayr. Exudyn github repository. https://github.com/jgerstmayr/EXUDYN (accessed on August 16, 2022), 2022.

[5] J. Gerstmayr and H. Irschik. On the correct representation of bending and axial deformation in the absolute nodal coordinate formulation with an elastic line approach. *Journal of Sound and Vibration*, 318(3):461–487, 2008.

[6] S. Holzinger and J. Gerstmayr. Time integration of rigid bodies modelled with three rotation parameters. *Multibody System Dynamics*, 53(4):345–378, 2021.

[7] W. Jakob, J. Rhinelander, and D. Moldovan. pybind11 – Seamless operability between C++11 and Python, 2016. https://github.com/pybind/pybind11.

[8] S. K. Lam, A. Pitrou, and S. Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery.

[9] P. Masarati, M. Morandini, and P. Mantegazza. An Efficient Formulation for General-Purpose Multibody/Multiphysics Analysis. *Journal of Computational and Nonlinear Dynamics*, 9(4), 07 2014. 041001.

[10] H. Mazhar, T. Heyn, A. Pazouki, D. Melanz, A. Seidl, A. Bartholomew, A. Tasora, and D. Negrut. CHRONO: A parallel multi-physics library for rigid-body, flexible-body, and fluid dynamics. *Mechanical Sciences*, 4:49–64, 02 2013.

[11] A. Pechstein and J. Gerstmayr. A Lagrange-Eulerian formulation of an axially moving beam based on the absolute nodal coordinate formulation. *Multibody System Dynamics*, 30(3):343–358, 2013.

[12] M. Pieber and J. Gerstmayr. A reconfigurable robot based on tetrahedral cells with adaptive shape. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2017.

[13] J. Schöberl. NETGEN An advancing front 2D/3D-mesh generator based on abstract rules. *Computing and Visualization in Science*, 1:41–52, 1997.

[14] J. Schöberl. C++11 Implementation of Finite Elements in NGSolve, ASC Report 30/2014. https://www.asc.tuwien.ac.at/~schoeberl/wiki/publications/ngs-cpp11.pdf (accessed on August 16, 2022), 2014.

[15] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[16] A. Zwölfer and J. Gerstmayr. The nodal-based floating frame of reference formulation with modal reduction. *Acta Mechanica*, 232(3):835–851, 2021.